

NMX
Message Exchange
Reference Guide

nilsen elektronikk as

Jan Erik Nilsen

July 26. 1999

nilsen elektronikk as

Gml. Drammensvei 12B

N-1369 Stabekk

Norway

Tel: +47 67 58 31 62

Fax: +47 67 58 97 61

<http://www.nilsenelektronikk.no>

TABLE OF CONTENTS

LICENSE AGREEMENT / DISCLAIMER.....	3
INTRODUCTION	4
REQUIREMENTS	4
OVERVIEW.....	4
HOW IT WORKS	5
BUFFER HANDLING.....	6
MESSAGE PRIORITY	6
DATA LARGER THAN A PACKET	6
DEADLOCK.....	7
TABLES AND INITIALISATIONS.....	7
WHAT IS A PORT?	9
PACKETS.....	10
HOW TO TRANSMIT A PACKET	11
HOW TO RECEIVE A PACKET	11
APPENDIX: HOW TO PREPARE PROC FOR NMX.....	12

License agreement / Disclaimer

Copyright © nilsen elektronikk as

NMX Message Exchange is a part of proc RTOS.

proc/nmx with documentation are properties of **nilsen elektronikk as**, Norway. **proc/nmx** is free software; you can use it, redistribute it and/or modify it under the terms below. By using, changing or redistributing the software, you accept the conditions below:

1. You are not allowed to remove or modify this copyright notice and License paragraphs, even if parts of the software is used.
2. The improvements and/or extentions you make **shall** be available for the community under **this** license, source code included. Improvements or extentions, including adaptions to new architectures, **shall** be reported and transmitted to Nilsen Elektronikk AS.
3. You must cause the modified files to carry prominent notices stating that you changed the files, what you did and the date of changes.
4. You may **not** distribute this software under another license without explicit permission from Nilsen Elektronikk AS, Norway.
5. This software is free, and distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. You **SHALL NOT** use this software unless you accept to carry all risk and cost of defects or limitations.

The software is provided «as is» and without any express or implied warranties, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose. The software should not be used within Life Support Systems. Life Support Systems are equipment intended to support or sustain life and whose failure to perform properly used in accordance with instructions provided can be reasonably expected to result in significant personal injury or death.

Introduction

This document describes the **NMX** Message Exchange (or Message Router). **NMX** is designed to use **proc** Real-Time Kernel, another product from nilsen elektronik as.

NB: See Appendix note on how to prepare **proc** for **NMX**. Also see **proc Real-Time Kernel Reference Guide**. For those who wish to do so, it should be easy to port to other RTOS'.

Requirements

The Message Exchange

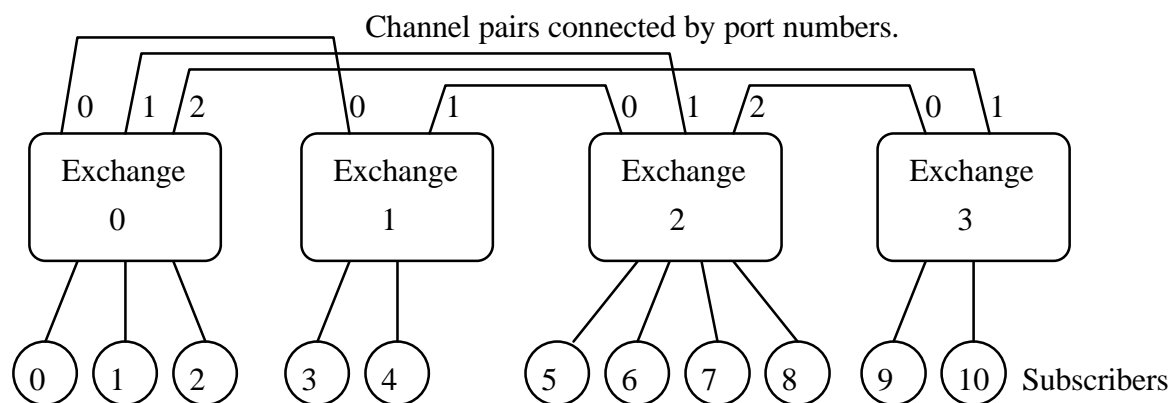
- is hardware and platform independent. It avoids use of architecture specific and other exclusive features.
- is symmetrical and compact.
- is easy to configure.
- is run-time re-configurable.
- accepts configuration tables in ROM or RAM.
- is impossible to dead-lock.

Overview

The message exchange provides an effective, flexible, convenient and robust way to pass messages between processes. Much like telephone subscribers, different processes can connect to an exchange. There is one exchange per physical processor. Different exchanges are connected in a network much like telephone exchanges.

In the local domain, i.e. within an exchange, messages are passed by pointers. This gives high throughput.

The figure below is an example of a exchange system. This configuration is also used in later examples.



How it works

A subscriber is a process, associated with a subscriber id. When a given process has data to pass to another process, it does:

- Allocates a packet, by pointer, from the exchanges packet pool.
- Writes source and destination id. to the header of the packet.
- Fills the packet with data.
- Writes data length to the header of the packet.
- Passes the packet, by pointer, to the exchange.

Then the exchange takes over. The following is done by an exchange function executed in the work space of the message source process:

- The destination id. is checked. The exchanges configuration tables give a port number by destination id. One dedicated port number means the destination is local.
- If the destination is local, the message pointer is passed to the destination process.
- If the destination is at a remote exchange the message is transmitted, byte by byte to the output port. Then the packet is recycled.

If the destination is local, the following is done:

- The destination process is waiting for a packet or the packet is waiting in a packet receive queue to be grabbed. The length of this queue can be different for different subscribers.
- The packet source and destination id. as well as the data length is read from the header of the packet.
- The packet data is copied to its final destination.
- The packet is recycled.

If the destination is remote, the following is done by the transmitting exchange:

- The transmitting process must reserve the output channel port first. This to avoid message jam, because more than one local process may start transmission simultaneously. The exchange has one semaphore per port for this purpose.
- Then the message header is transmitted. The header contains source and destination id. as well as data length.
- Then, if length is greater than zero, the message body is transmitted.
- Finally, the port is released.

The receiving exchange has one process per port waiting for data from a input channel port. The following happens:

- The receiving process allocates a packet from the exchanges packet pool and waits for data to arrive.

- First, the message header arrives with source and destination id. as well as data length. This data is written to the packet header.
- Then the given number of data is transferred from the input channel to the packet body.
- The complete packet is then handled same as a packet from a local subscriber. It can be passed to a local subscriber or re-transmitted via another port.

Buffer handling

There are buffers at different levels. The packet pool contains free packets available for any subscriber or receiving process of the exchange. If the packet pool is emptied, allocating processes are blocked until packets are recycled by other processes.

A packet is passed by pointer to an output channel port or to the input queue of a receiving process. A channel may be buffered or not. By using buffer space for one packet, better performance is gained.

The input queue of a receiving process has a maximum space given by configuration. If a small buffer space is used, the packet pool is saved from being exhausted, but the entire message path can be congested, effectively blocking the traffic in the involved channels. If a larger buffer is used, the packet pool can be emptied, but the data path saved from congestion. However, an empty packet pool will block the entire exchange and may cause congestion's in other part of the system.

Some messages are important, and should not be lost. In other cases, it may be better to lose messages than have the exchange system blocked. This can be individual configured.

Message priority

Important messages can be routed via channels dedicated for prioritised messages. A such message will not be buffered together with other messages.

Data larger than a packet

Data larger than a packet must be chopped and transferred in parts. This is not a part of the message exchange and can be handled at a higher level.

Deadlock

The necessary four conditions for deadlock must all be true:

1. Mutual exclusion. A necessary recourse can not be shared.
2. Hold and wait. One process is holding a resource and waiting to acquire additional resources held by other processes.
3. No pre-emption. Resources can only be released voluntarily.
4. Circular wait. Processes are mutual waiting for resources held by other processes.

Mutual exclusion: This applies for an output channel, the last packet in the packet pool and the last space of a receiving queue.

Hold and wait: After allocating the last free packet, the process is waiting for an output channel or space in a receiving queue.

No pre-emption: Always true.

Circular wait: The order in which resources are allocated is always the same. Therefore, this condition is always false.

The message exchange system should never be accused of causing deadlocks.

Tables and initialisations

The following types defines the tables used for configuration:

The eXchange Configuration Struct holds information for one exchange. There are one such struct per exchange:

```
typedef struct {          /* exchange configuration struct */
    int    pnum;          /* number of external ports      */
    int    lsnum;         /* number of local subscribers   */
    int    tsnum;         /* total number of subscribers   */
    int    bufnum;        /* total number of packets       */
    int    bufsiz;        /* packet net size in bytes      */
} XCS;
```

Every exchange in the system must have an identical array of eXchange Subscriber Description Structs. The number of elements is XCS.tnum.

```
typedef struct {          /* subscriber description struct */
    int    id;           /* subscriber identity number */
    int    attr;        /* subscription options */
    int    bufmax;      /* input queue maximum size */
} XSDS;
```

Every exchange in the system must have an individual array of eXchange Port/Channel assignments. If the system is only one exchange, it has zero external ports (XCS.pnum) and a NULL pointer can be used. The first channel pointer is for data from the outside world to toe exchange, and the second is for data from the exchange to the outside world.

```
typedef struct {          /* port/channel assignment */
    CHAN   *wxcp;        /* world/exchange channel ptr */
    CHAN   *xwcp;        /* exchange/world channel ptr */
} XPC;
```

Configuration is explained with an example. See the figure under section «Overview». This example shows the configuration tables of Exchange 0.

```
XCS xcsa = {
    3,          /* number of external ports */
    3,          /* number of local subscribers */
    11,        /* total number of subscribers */
    16,        /* total number of rx buffers */
    256 };     /* buffer size in bytes */
```

```
int gsa[11] = {
    /* subscriber 0 at port */ NMX_LOCAL,
    /* subscriber 1 at port */ NMX_LOCAL,
    /* subscriber 2 at port */ NMX_LOCAL,
    /* subscriber 3 at port */ 0,
    /* subscriber 4 at port */ 0,
    /* subscriber 5 at port */ 1,
    /* subscriber 6 at port */ 1,
    /* subscriber 7 at port */ 1,
    /* subscriber 8 at port */ 1,
    /* subscriber 9 at port */ 2,
    /* subscriber 10 at port */ 2  };
```

```
XSDS lsa[3] = {
    { 0, 0,      5 },      /* id, attr, bufmax */
    { 1, NMX_SANB, 4 },
    { 2, 0,      3 } };
```

```
CHAN ichan[3];
CHAN ochan[3];
```

```
XPC pca[3] = {
    { &ichan[0], &ochan[0] },
```

```
{ &ichan[1], &ochan[1] },  
{ &ichan[2], &ochan[2] } };
```

```
XHANDLE *xha;
```

```
void openExchange_A(void)  
{  
    xha = msgExchangeInit(&xcsa, &gsa[0], &lsa, &pca);  
}
```

Some symbols are used here:

NMX_LOCAL The port number for local subscribers.
NMX_SANB Recycle oldest message if the receive queue is full. No block.

The tables are read only, and can reside in ROM.

NB: Before opening exchanges, all channels must be initialised. This is not shown here.

The function `msgExchangeInit()` allocates memory and initialises internal structs. Process id. and workspace is allocated for the port processes, in this case 3. A packet pool with 16 packets is allocated, each with net data size 256. Other memory are allocated as well, and the whole environment is contained by a XHANDLE struct.

The allocated memory is free by the function `msgExchangeFree()`.

The exchange processes are started by the function `msgExchangeOpen()` and terminated by the function `msgExchangeClose()`.

What is a port?

A port is a bitwise channel pair. One for receiving data and another for transmitting data. The data type CHAN describes a handle for a channel. A such channel must block when it is full or empty. More information about the CHAN type and functions can be found in the **proc Real-Time Kernel Reference Guide**.

Two channel pairs can be connected via e.g. UARTs. This way, two exchanges can be connected.

The exchange formats a packet into a stream of bytes of format:

bytes	2	2	2	<len>
	<src>	<dst>	<len>	[<data>]

<src> Subscriber Id. of message source.
 <dst> Subscriber Id. of message destination.
 <len> Length of <data> in bytes. Range: 0..(XCS.bufsiz - 1)
 <data> Message body.

Packets

The PACKET type is defined in **proc Real-Time Kernel** «packet.h» (NB: See appendix note):

```
struct packet {
    struct packet    *next;        /* ptr to next          */
    struct packetpool *ppool;     /* ptr to home pool    */
    int              dst;         /* packet destination  */
    int              src;         /* packet source       */
    int              length;      /* data length         */
    char             data[2];     /* data buffer         */
};
typedef struct packet PACKET;
```

The two pointers must not be altered. «next» is used for linking in the pool or packet lists. «ppool» is a pointer to its home pool. A recycled packet will always land in the pool it belongs to. This is because there can be many different packet pools with different packet types and sizes.

«dst», «src», «length» and «data» are to be altered by the user. «data» has a space of 2, but don't be confused by that, because the struct is flexible at its end.

More information about the PACKET type and functions can be found in the **proc Real-Time Kernel Reference Guide**.

How to transmit a packet

Allocate a packet from the exchanges packet pool.

```
PACKET *pp;  
XHANDLE *xp;  
  
pp = msgPacketAlloc(xp);
```

Write source and destination id., message data length and message body.

```
pp->dst      = destination_id;  
pp->src      = my_id;  
pp->length   = data_length;  
memcpy(&pp->data[0], &buffer[0], data_length);
```

Pass the packet to the exchange.

```
msgPacketPut(pp, xp);
```

How to receive a packet

It is easy to receive a packet:

```
PACKET *pp;  
XHANDLE *xp;  
  
pp = msgPacketGet(my_id, xp);
```

If blocking must be avoided, just check the input queue first:

```
number_of_waiting_packets = msgPacketsWaiting(my_id, xp);
```

Get source and destination id., message data length and message body:

```
destination_id = pp->dst; /* same as my_id */  
remote_id      = pp->src;  
data_length    = pp->length;  
memcpy(&buffer[0], &pp->data[0], data_length);
```

Finally free the packet:

```
packetFree(pp, xp);
```

Appendix: How to prepare proc for NMX

proc Real-Time Kernel is another product from nilsen elektronikk as. The PACKET type must be changed a little bit, and the **proc** library re-build.

The original PACKET is defined:

```
struct packet {
    struct packet    *next;        /* ptr to next          */
    struct packetpool *ppool;     /* ptr to home pool    */
    int              length;      /* data length         */
    char             data[2];     /* data buffer         */
};
```

Destination and source is missing. **NMX** needs this PACKET type:

```
struct packet {
    struct packet    *next;        /* ptr to next          */
    struct packetpool *ppool;     /* ptr to home pool    */
    int              dst;         /* packet destination  */
    int              src;         /* packet source       */
    int              length;      /* data length         */
    char             data[2];     /* data buffer         */
};
```

The changes are made in «packet.h». The proc library needs to be compiled and re-build.